YEI Prio API Quick Start Guide

Purpose

The purpose of this document is to introduce users to the YEI Prio Application Programming Interface (API). After reading this document users will be able to instantiate the various classes of the API to interact with the Prio devices and interpret any read data in a meaningful way.

Overview

The Prio API is a collection of convenience functions that wrap all normal functionality of YEI Technology's family of Prio devices for use in a program written in C/C++ or any language that can import compiled libraries (.dll, .so, etc).

Software Requirements

Operating System – Windows 7 32-Bit/64-Bit
(*Windows XP 32-Bit/64-Bit*, *Windows Vista 32-Bit/64-Bit*, *Windows 8*, and *Unix* are untested)

Getting Started with the Prio API

When connecting to and communicating with a Prio device via the API, there are a few common steps that must be taken before full communication may take place. These steps are as follows:

1. Link to the API library.
2. Include the API header in your file.
3. Know the serial port of the Prio device.
4. Construct an interface instance of the Prio device.

If guidance is required for linking the library to the user's development environment, please refer to one of the following Quick Starts:

YEI Prio API Quick Start Guide: Setup Environment for Visual Studio

The next section will go over the last 3 steps using code snippets from the example *creating_instances.c* and explain some key components in using the API. For a more detailed description of the API and its components, please refer to the *YEI Prio API User's Manual* or the documentation file for the Prio API.

Creating a Basic C Code Example

The following examples can be found on the Prio website (www.priovr.com).

Creating Instances

This example shows how to use the API to connect to and find a Prio device. When a Prio device is connected a *prio_device_id* is returned, which is needed to call many functions in the API.

```
#include <stdio.h>
#include <stdint.h>
#include "prio_api.h"

int main()
{
    prio_device_id device;
    PRIO_ERROR error;
```

If the serial port is already known for the Prio device, we can create an instance without doing a search.

```
    const char* com_port_str = "COM7";    // <-- Edit this to your device's serial port
    printf("====Creating Prio Device on %s====\n", com_port_str);
    device = prio_createDeviceStr(com_port_str, PRIO_TIMESTAMP_SYSTEM);
```

It is always best to check if the *prio_device_id* is valid. This is usually the case unless the Prio device's serial port is already in use.

```
    if (device == PRIO_NO_DEVICE_ID)
    {
        printf("Failed to communicate on %s\n", com_port_str);
    }
    else
    {
        printf("Created a Prio device on %s\n", com_port_str);
        prio_destroyDevice(device);
    }
    printf("================================\n");
```

If the serial port is not known for the Prio device, we can do a search for the device. We can do this by calling the *prio_getComPorts* function. Many functions of the API take a structure or an array to get the data. This particular function can take an array of the *prio_ComPort* structure, the size of the array, an offset in the array, and a filter parameter of *PRIO_FIND*. Then it fills the array with *prio_ComPort* structures based on the filter. The filter can be used to find unknown, specific, or all connected Prio devices. Each *prio_ComPort* structure contains a *char* array for the serial port name and a *PRIO_TYPE* that denotes the type of the device for the serial port.

```
printf("====Creating Prio Devices from Search====\n");
prio_ComPort com_ports[20];
int device_count = prio_getComPorts(com_ports, 20, 0, PRIO_FIND_ALL_KNOWN);
printf("====Found %d Devices====\n", device_count);
for  (int i = 0; i < device_count; ++i)
{
    device = prio_createDeviceStr(com_ports[i].com_port, PRIO_TIMESTAMP_SYSTEM);
    if (device == PRIO_NO_DEVICE_ID)
    {
         printf("Failed to create a device on %s\n", com_ports[i].com_port);
    }
    else
    {
        printf("Created a Prio device on %s\n", com_ports[i].com_port);
        prio_destroyDevice(device);
    }
}
printf("================================\n");
```

If the filter parameter for *prio_getComPorts* is set to also find unknown types, we must be careful in trying to create a *prio_device_id* as commands are sent over the serial port which could make devices other than Prio devices act in an undesirable way. So be sure to know what port your Prio device(s) are on when creating instances.

We can get more information about a Prio device using the *prio_DeviceInfo* structure and the *prio_getDeviceInfo* function. The *prio_DeviceInfo* structure contains a *PRIO_TYPE* that denotes the type of the device, a *uint32_t* that denotes the serial number for the device, a *char* array that denotes the firmware version of the device, a *char* array that denotes the hardware version of the device, and a *PRIO_FIRMWARE_COMPATIBILITY* that denotes the compatibility level of the firmware.

```
printf("====Checking Unknown Ports for Prio Devices====\n");
device_count = prio_getComPorts(com_ports, 20, 0, PRIO_FIND_UNKNOWN);
printf("====Found %d Devices====\n", device_count);
prio_DeviceInfo device_info;
for (int i = 0; i < device_count; ++i)
{
    device = prio_createDeviceStr(com_ports[i].comport, PRIO_TIMESTAMP_SYSTEM);
    if (device == PRIO_NO_DEVICE_ID)
    {
         printf("Failed to create a device on %s\n", com_ports[i].com_port);
    }
    else
    {
        printf("Created a Prio device on %s\n", com_ports[i].com_port);
```

It is also always best to check if the command succeeded. This is very helpful to debug what went wrong especially when communicating wirelessly.

```
            error = prio_getDeviceInfo(device, &device_info);
            if (error == PRIO_NO_ERROR)
            {
                printf("============(%s)=============\n", com_ports[i].com_port);
                printf("DeviceType:%s\n", prio_type_string[device_info.device_type]);
                printf("Serial:%08X\n", device_info.serial_number);
                printf("HardwareVersion:%s\n", device_info.hardware_version);
                printf("FirmwareVersion:%s\n", device_info.firmware_version);
                printf("Compatibility:%d\n", device_info.fw_compatibility);

            }
            else
            {
                printf("ERROR: %s\n", prio_error_string[error]);
            }
            prio_destroyDevice(device);
        }
    }
    printf("================================\n");
```

Finally before quitting from your code, it is best to call the *prio_resetPrioApi* function to properly clean-up the memory the API has allocated. This also helps assure the API will work properly the next time it is used.

```
    error = prio_resetPrioApi();
    if (error != PRIO_NO_ERROR)
    {
        printf("ERROR: %s\n", prio_error_string[error]);
    }

    printf("\nFinished press Enter to continue");
    getchar();
    return 0;
}
```

Getting Information

This example demonstrates getting basic data from the sensor. As shown in the previous example a connection to the Prio device must be made.

```
#include <stdio.h>
#include "prio_base_api.h"

int main(){
    prio_ComPort comport;
    prio_device_id prio_device;
    PRIO_ERROR error;

    if (prio_getComPorts(&comport, 1, 0, PRIO_FIND_BS))
    {
        prio_device = prio_createDeviceStr(comport.com_port, PRIO_TIMESTAMP_SYSTEM);
        if (prio_device == PRIO_NO_DEVICE_ID)
        {
            printf("\tERROR: Failed to create a sensor on %s\n", comport.com_port);
            return 1;
        }
    }
    else
    {
        printf("\tNo sensors found\n");
        return 1;
    }
    printf("Enumerating the Prio HUB\n");
    error = prio_enumerateHub(prio_device, nullptr);

    printf("Finding all present nodes after enumeration.\n");
    uint8_t active_nodes_len = 0;
    for (i = 0; i < sizeof(present_nodes) / sizeof(uint8_t); i++)
    {
        uint8_t present;
        error = prio_isSensorPresent(prio_device, i, &present);
        present_nodes[i] = present;
        if (present)
        {
            printf("\tSensor Node %d is present\n", i);
            active_nodes_len++;
        }
    }
```

Once the enumeration process is complete data can be retrieved. Many functions provide a structure or an array to get the data; *prio_getTaredOrientationAsQuaternion* writes out a quaternion as an array of four floats along with an optional time stamp as an unsigned integer.

```
    printf("Get the Tared Orientation of each present node.\n");
    float quat[4];
    uint32_t timestamp;
    for (i = 0; i < sizeof(present_nodes) / sizeof(uint8_t); i++)
    {
        if (present_nodes[i])
        {
            error = prio_getTaredOrientationAsQuaternion(prio_device, i, quat, &timestamp);
            if (!error)
            {
                printf("\t%d Quat: %f,%f,%f,%f-Timestamp: %u\n",
                                        i, quat[0], quat[1], quat[2], quat[3], timestamp);
            }
            else
            {
                printf("ERROR: %s\n", prio_error_string[error]);
            }
        }
    }
```

After getting the data needed the example destroys the connection with the Prio device then resets the API.

```
    prio_destroyDevice(device);

    error = prio_resetPrioApi();
    if (error != PRIO_NO_ERROR)
    {
        printf("ERROR: %s\n", prio_error_string[error]);
    }

    printf("\nFinished press Enter to continue");
    getchar();
    return 0;
}
```

Streaming Data with the Prio API

Other than using write/read commands to get data from a Prio device, the Prio API is able to make use of the Prio device's streaming option. It does so by using C++11's multithreading library. Streaming data is the best option for real-time applications.

Setting the Streaming Slots

The Prio devices allows users to choose up to 8 different types of data they want to stream by setting the streaming slots of the device. The API makes this easy by providing a *PRIO_STREAM_COMMAND* enum type that denotes what commands can be streamed and a *prio_stream_commands* type that the function *prio_setStreamingSlots* takes as a parameter. By default the streaming slots are set to only stream the tared orientation as a quaternion from the sensor nodes on creation.

```
uint32_t timestamp;
prio_stream_commands prio_slots[8] =
{
    PRIO_GET_TARED_ORIENTATION_AS_QUATERNION,
    PRIO_GET_CORRECTED_ACCELEROMETER_VECTOR,
    PRIO_NULL,
    PRIO_NULL,
    PRIO_NULL,
    PRIO_NULL,
    PRIO_NULL,
    PRIO_NULL
};

error = prio_setStreamingSlots(prio_device, prio_slots, &timestamp);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Getting the Size of the Stream Data

Once the streaming slots are set to return the desired data, we need to know the full size of the stream data. To start, we can calculate or get the size of the data denoted by the streaming slots. In the previous example, the streaming slots are set up to stream the tared orientation as a quaternion and corrected accelerometer data from the sensor nodes. Now note that each stream command is a different number of floats. From there we can calculate the size, as it would be 4 floats for the quaternion and 3 floats for the accelerometer data, making it a total of 7 floats. Now the API provides the function *prio_getLengthOfStreamData* that requires less knowledge of the streaming slots.

```
uint32_t stream_data_len = 0;
error = prio_getLengthOfStreamData(prio_device, &stream_data_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Next, we need to get the number of active sensor nodes from the Prio device. This can be done using the *prio_getActiveSensors* function.

```
uint8_t* active_nodes;
uint8_t active_nodes_len = 0;
error = prio_getActiveSensors(prio_device, NULL, 0, &active_nodes_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
active_nodes = (uint8_t*)malloc(active_nodes_len * sizeof(uint8_t));
error = prio_getActiveSensors(prio_device, active_nodes, active_nodes_len, NULL);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Finally, we multiply the numbers together to get the full length of the stream data and we can now be sure we have allocated enough memory for getting the stream data. The API also provides another way of getting the full length of the stream data with the function *prio_getFullLengthOfStreamData*.

```
uint32_t full_stream_data_len = 0;
error = prio_getFullLengthOfStreamData(prio_device, &full_stream_data_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Now that we know how much memory needs to be allocated we can start streaming data by calling *prio_startStreaming*. The API gives a few methods to retrieve the stream data: non-blocking, blocking, and recorded. Each method is designed for an array of different applications in mind. For convenience all methods of retrieving data also write the header data out to a *prio_StreamHeaderData* structure parameter.

```
prio_StreamHeaderData header_data;
float* stream_data;
uint32_t stream_timestamp;
stream_data = (float*)malloc(full_stream_data_len * sizeof(float));

error = prio_startStreaming(prio_device, 0, NULL);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Getting the Stream Data: Non-blocking

Using the non-blocking method, *prio_getLastStreamData*, is ideal for real-time applications as it returns information the fastest.

```
// In some kind of loop
error = prio_getLastStreamData(prio_device, &header_data, stream_data,
                               full_stream_data_len, &stream_timestamp);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Getting the Stream Data: Blocking

Using the blocking method, *prio_getLatestStreamData*, is ideal for applications wanting unique data such as when data logging. The function accomplishes this by taking a timeout parameter in milliseconds to know how long to wait for new data to arrive.

```
// In some kind of loop
error = prio_getLastStreamData(prio_device, &header_data, stream_data,
                               full_stream_data_len, 16, &stream_timestamp);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Getting the Stream Data: Recorded

Now with getting recorded stream data, we should set a sample size for the recording. By default it is set to the max size of an std::deque container, but there is no guarantee that it would not fail before reaching that size. So it is best to set a max sample size of the recording using the *prio_setMaxRecordedSamples* function. Note that this container is a First-In-First-Out (FIFO) data structure, and once the max size is reached the oldest data in the container will be removed.

```
uint32_t max_samples_len;
...
error = prio_setMaxRecordedSamples(prio_device, max_samples_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Once the max sample size is set, we need to tell the API to start recording. This can be done in 2 ways. One way, is setting the start_recording parameter for the *prio_startStreaming* function to some value greater than 0. The other is to call the *prio_startRecordingData* function after streaming has already started.

Now we must decide what method we wish to use to get the recorded samples. The API provides 3 methods of getting recorded samples with one of them requiring an addition setup before calling. One method is to use the *prio_popFrontRecordedSample* function. This function removes and returns the recorded sample at the front of the recording container.

```
// In some kind of loop
error = prio_popFrontRecordedSample(prio_device, &header_data, stream_data,
                                    full_stream_data_len, &stream_timestamp);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Another method is to use the *prio_getRecordedSampleAtIndex* function. This function returns the recorded sample at the given index from the recording container.

```
uint32_t idx;
...
// In some kind of loop
error = prio_getRecordedSampleAtIndex(prio_device, &header_data, stream_data,
                                      full_stream_data_len, idx, &stream_timestamp);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

The last method available to use the prio_getRecordedSamples function. This function returns all of the recorded samples in the recording container. With this method, a setup of getting the size of the recorded container using prio_getLengthOfRecordedSamples and prio_getFullLengthOfRecordedSamples, and stopping the recording using prio_stopRecordingData is necessary. Note that allocating memory for the header data and timestamps is only necessary if that information is wanted.

```
error = prio_stopRecordingData(prio_device);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}

uint32_t recorded_samples_len = 0;
error = prio_getLengthOfRecordedSamples(prio_device, &recorded_samples_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}

uint32_t full_recorded_samples_len = 0;
error = prio_getFullLengthOfRecordedSamples(prio_device,
                                            &full_recorded_samples_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}

prio_StreamHeaderData* recorded_header_data;
float* recorded_data;
uint32_t* recorded_timestamps;
recorded_header_data = (prio_StreamHeaderData*)malloc(
    recorded_samples_len * sizeof(prio_StreamHeaderData));
recorded_data = (float*)malloc(full_recorded_samples_len * sizeof(float));
recorded_timestamps = (uint32_t*)malloc(recorded_samples_len * sizeof(uint32_t));

error = prio_getRecordedSamples(prio_device,
                                &recorded_header_data, recorded_samples_len,
                                recorded_data, full_recorded_samples_len,
                                &recorded_timestamps, recorded_samples_len);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Stopping Streaming and Clean Up

To stop streaming call the *prio_stopStreaming* function. This will also stop recording if it is still recording. Once streaming has stopped, it is best to clean up any allocated memory we no longer need. To clean up any allocated memory the API used for recording samples, call the *prio_clearRecordedSamples* function.

Now that you are more familiar with the functionality of the Prio API and how to connect to and find a Prio device, you can begin creating your own examples and applications using the API.

YEI Technology

630 Second Street
Portsmouth, Ohio 45662

Toll-Free: 888-395-9029
Phone: 740-355-9029

www.YeiTechnology.com
www.PrioVR.com

©2007-2014 YEI Corporation
Printed in USA